Mobile Agent Security

Levent Ertaul
Department of Mathematics and Computer Science
California State University, East Bay,
Hayward, CA, USA.
levent.ertaul@csueastbay.edu

Jayalalitha Panda
Department of Mathematics and Computer Science,
California State University, East Bay,
Hayward, CA, USA
jaya.panda@gmail.com

Abstract — *Mobile agent technology is a new paradigm of distributed computing that can replace the conventional client-server model. However, it has not become popular due to some problems such as security. Threats to mobile agent security generally fall into three main classes: disclosure of information, denial of service, and corruption of information. Four threat categories are identified: threats stemming from an agent attacking an agent platform, an agent platform attacking an agent, an agent attacking another agent on the agent platform, and other entities attacking the agent system. In this paper, we discuss our implementation of two of the security approaches called Mixed Multiplicative Homomorphic Encryption scheme and Secure Dynamic Programming. These security approaches protect the mobile agents from malicious agent platforms. We also discuss our agent integrity checking mechanism that is implemented using SHA1 digest algorithm. These implementations are done in the IBM's JAVA Mobile agent system called Aglets and provide Confidentiality and Integrity services to the mobile agents.*

Keywords: Aglet, Homomorphic Functions, Mobile Agents, Mobile Cryptography, Dynamic Programming.

1. Introduction

A mobile agent is a software object that is not bound to the system where it begins its execution. It has the unique ability to transport itself from one system in a network to another. The ability to travel allows a mobile agent to move to a system that contains an object with which the agent wants to interact and then to take advantage of being in the same host or network as the object [1].

Mobile agents reduce network traffic, overcome network latency, encapsulate protocols, execute asynchronously and autonomously, adapt dynamically, naturally heterogeneous and are robust and fault-tolerant [1].

We have chosen the IBM's JAVA Mobile Agent System – AGLETS for our implementation. We chose a JAVA-based mobile agent system because of the following benefits [1]: *Platform independence*:, *Secure Execution*, *Dynamic Class Loading*:, *Multithread Programming*, *Object Serialization*, *Reflection.*

There are a few other interesting JAVA-based mobile agent systems which are Odyssey, Concordia, and Voyager [1].

All these agent systems need some security and that is why we decided to implement two of the available security models that can counteract some of the security attacks that have been described in section II.

The rest of the paper is organized as follows: In Section II, we discuss about security threats in mobile agents. In Section III, a brief introduction to the IBM Aglets is given. In Section IV, we discuss about some of the agent security models. In Section V, we talk about the Mixed-Multiplicative Homomorphic encryption (MMH) scheme. In Section VI, MMH cryptosystem is detailed with an example. Section VII explains our implementation of MMH cryptosystem in IBM Aglets. Section VIII, details the secure dynamic programming protocol and Section IX explains our implementation of that protocol in IBM Aglets. In Section X we explain our implementation of integrity checking mechanism in IBM Aglets.

2. Security Threats in Mobile Agents

Mobile agents moving around the network are not safe. The Agent-to-Host, Agent-to-Agent, Host-to-Agent, Other-to-Agent Host attacks are the kinds of security attacks that are possible in a Mobile Agent System [2]:

We have implemented security systems that protect the agents from the attacks by the malicious hosts.

A malicious host can be defined in a general way as a party that is able to execute an agent that belongs to another party and tries to attack the agent in some way [3].

It seems obvious that if the remote host is to execute a process, the process can have no secrets from that host.

In the next section, we discuss about some specifics pertaining to IBM's JAVA Mobile Agent System – AGLETS in which we have implemented our security and integrity approaches.

3. Mobile JAVA Agent: The Aglet model

Aglet is a JAVA based mobile agent system developed by IBM [1].

Agents --- which are called aglets in this system --- migrate between agent servers (called aglet contexts) located on different network hosts. A distinguishing feature of Aglets is its callback-based programming model [1, 4].

Aglets are hosted by an Aglet server in a way similar to the way applets are hosted by a web browser. The Aglet server provides an environment for aglets to execute in, and the JAVA virtual machine and the Aglet security manager make it safe to receive and host aglets.

Now let us give an introduction to the aglet object model. This model was designed to benefit from the agent characteristics of JAVA while overcoming some of the deficiencies in the language system. In the aglet object model, a mobile agent is a mobile object that has its own thread of control, is event-driven, and communicates by message passing.

Now let us take a closer look at the model underlying the

Aglet API. This model defines a set of abstractions and the behavior needed to leverage mobile agent technology in internet-like, open wide-area networks: The key abstractions are aglet, proxy, context and identifier [1].

The following list summarizes the fundamental operations of an aglet: *creation, cloning, dispatching, retraction, deactivation, activation, and disposal* [1].

In the next section we discuss about some of the Agent security models.

4. Agent Security Models

The following are some of the available security models for mobile agents. Each one of them can be used to provide security for different applications.

4.1 Computing with Encrypted Data

Encrypted programs can be used to protect agents from malicious hosts. Encrypted programs are programs that consist of operations that work on encrypted data. Agents are produced by converting an agent specification into some executable code plus initial, encrypted data. Since, the attacker cannot break the encryption of the data it cannot read or manipulate the original data.

The problem of computing with encrypted data has been described in [6] in the following way:

Bob has an algorithm to compute function f and is willing to compute f(x) for Alice. Alice wants to compute f on her private input x but does not want to reveal x to Bob. Furthermore Alice should not learn anything substantial about the algorithm of Bob for computing f.

The solution proposed in [7] yields a highly interactive protocol to this problem of the model of “Boolean circuits”; it allows Alice to encrypt the input data x in such a way that Bob can compute $f(x)$ for her without getting to know the clear text x.

4.2 Computing with Encrypted Functions

The mobile code can not be effectively protected against the executing system because the host has full control over its execution and it may potentially fully understand the code and eventually can change it in any way it wants. But the argument is that we can obtain a system where a host can execute an encrypted function without having to decrypt it. Thus, functions would be encrypted such that the resulting transformation can be implemented as a (mobile) program that will be executed on a remote host. The executing computer will see the program's clear text instructions but will not be able to understand the function that the program implements. [5].

The problem of computing with encrypted functions has been described in [6] in the following way:

Alice has an algorithm to compute a function f. Bob has an input x and is willing to compute f(x) for her, but Alice wants Bob to learn nothing substantial about f. Moreover, Bob should not need to interact with Alice during the computation of f(x).

Privacy Homomorphisms (PHs) that were formally introduced in [7] are basically encrypted functions. The security gain of privacy homomorphism is in a multilevel security environment: data can be encrypted at a classified level, be processed by an unclassified computing facility and the result be decrypted by the classified level [8].

4.3 Standard Cryptography

In standard cryptographic techniques the keys need to be kept secret and the processing needs to be done in a secure execution environment.

But, Mobile agents should be allowed to execute in untrusted or unsecure hosts and still have guarantees for their correct execution. Protection Mechanisms for Mobile Agents should be provably secure.

In the next section, we discuss about the Mixed-Multiplicative Homomorphic Encryption scheme.

5. Mixed-Multiplicative Homomorphic Encryption Scheme (MMH)

Here we discuss about the security approach presented in [9] which we implemented. This approach focuses on extending the mobile cryptography approach, proposed in [10, 5, 11], in terms of privacy and integrity, and explore its usefulness and effectiveness in protecting mobile agents. To extend mobile cryptography, in [5], composite functions and additive-multiplicative homomorphism are considered to encrypt mobile agents. Homomorphic Encryption Scheme (HES) enables direct computation on encrypted data without decryption.

Properties of HES that are needed to secure mobile agents are [10, 11]:

- *additively homomorphic:* computing $E(x+y)$ from $E(x)$ and $E(y)$ without revealing x and y
- *multiplicatively homomorphic:* computing $E(xy)$ from $E(x)$ and $E(y)$ without revealing x and y
- *mixed-multiplicatively homomorphic:* computing $E(xy)$ from $E(x)$ and y without revealing x.

The mobile agent encrypted with HES will be able to run on any host without decryption. Also, the HES encrypted agent will generate encrypted results, which will be decrypted by the agent owner. This will improve the overall security of the mobile agents. Computation on encrypted data protects the data from the untrusted hosts.

But, the challenge is to find encryption schemes for arbitrary functions. We can find encrypting transformations for specific function classes such as polynomials and rational functions [10].

Also, an important observation made in [5] is that for computing with encrypted polynomial it is not necessary to have both the additive and multiplicative property of an encrypted function: it is sufficient that the encryption supports addition and "mixed multiplication" [11].

The next section explains the MMH cryptosystem. The subsequent section details our implementation of this cryptosystem in IBM Aglets.

6. MMH (Mixed Multiplicative Homomorphic) Cryptosystem

MMH cryptosystem presented in [9] uses a large number, n, such that $n = p \times q$ where p and q are large prime numbers. Let $Z_p = \{ x \mid x \leq p\}$ be the set of original plaintext messages $Z_n = \{ x \mid x < n \}$ be the set of cipher text message and $Q_p = \{a \mid a$ *is not an element of* $Z_p\}$ be a set of encryption clues. The types of operations defined are addition and multiplication on Z_p.

The encryption and decryption algorithms are as follows:

Encryption: Given x is an element of Z_p, pick a random number a in Q_p such that $x = a \bmod p$. Compute the encrypted value $y = E_p(x) = a \bmod n$. (This can be accomplished by picking a random r and creating $a = x + rp$.)

Decryption: Given $y = E_p(x)$ is an element of Z_n , use the key p to recover $x = D_p(y) = y \bmod p$.

This cryptosystem is additively, multiplicatively, and mixed-multiplicatively homomorphic.

Example (Multiplication): Let $p = 17, q = 13, n = 221 = p \times q$ and the values, $x1 = 8$ *and* $E(8) = 59$ and $x2 = 2$ where $E(2) = 36.$

$(59 \times 36) \bmod 221 = 135$

Decrypting *135* yields,

$16 = 135 \bmod 17$

which is the same as the unencrypted multiplication result $x1 \times x2 = 8 \times 2 = 16.$

A mixed-multiplicative homomorphism allows encryption of a plaintext message without any knowledge of the cryptosystem including the keys and encryption algorithm. An advantage of this approach is that the encryption can be done in real-time, because the encryption of the plaintext, y, requires only a single invocation of the encryption function.

One possible application of the mixed-multiplicative homomorphic encryption scheme is multi-party computation, where each party does not want to reveal its data to the other participants. A mixed-multiplicative homomorphic encryption scheme will allow each participant to encrypt inputs to a program, and perform the direct computation on the encrypted data.

This scheme is protected against the ciphertext-only attack due to the difficulty in factoring of a large prime number. But, it needs to be protected against the following attacks [9]:

Known-Plaintext Attack: Cryptanalyst knows a plaintext-ciphertext pair (x, y). Since $y = E(x) = (x + rp) \bmod n$, $rp \bmod n = E(x) - x \bmod n$. So, p must be $gcd(rp, n)$.

Integrity Attack: Since decryption is performed modulo p, any unencrypted number $x < p$ will be deciphered as itself. So, an encrypted value can be replaced with a chosen value and claim it to be encrypted

Automatic encryption of Remote input: By definition of the MMH, the remote input x, can be automatically encrypted by a malicious host by multiplying x by $E(1)$ assuming if the agent owner provides $E(1)$. No need to know the encryption algorithm.

7. Implementation of MMH Cryptosystem in IBM aglets

Our implementation does the encryption and decryption functions of the MMH cryptosystem. There are two aglets in our implementation: The creator aglet *MMHEncrDecrAgletApp* whose class file is given in Table 1 and the proxy aglet, *MMHEncryptDecrypt* whose class file is given in Table 2. The proxy consists of the function that does the multiplication on encrypted data. The creator aglet encrypts the two integers *(x1 and x2)* whose multiplication operation is to be subcontracted to the host at the destination URL. The creator aglet then creates the aglet proxy, *MMHEncryptDecrypt* and passes the encrypted integers as arguments. The proxy is then dispatched to the destination whose URL is given. At the destination the multiplication is done on the encrypted integers. The creator aglet then collects the results by exchanging messages with the proxy. It then decrypts the results and prints it to the console. Finally the proxy is retrieved from the destination and disposed.

Table 1. Creator Aglet

```
public class MMHEncrDecrAgletApp extends Aglet {
        public void run() {
                BigInteger encResult;
                BigInteger p = new BigInteger("11");
                BigInteger q = new BigInteger("7");
                BigInteger n = new BigInteger("77");
                BigInteger x1 = new BigInteger("5");
                BigInteger x2 = new BigInteger("5");
                BigInteger r = new BigInteger("3");
/* a1 and a2 are the encrypted values of the integers x1 and x2  respectively.
The encryption is done using the MMH cryptosystem  */
                BigInteger a1 = x1.add((r.multiply(p)));
                BigInteger a2 = x2.add((r.multiply(p)));
                try {
                        Object args = new Object[ ] {
                                                a1,a2,n };
        URL destination = new URL("atp://LIFEBOOK:3000");
/*  The MMHEncryptDecrypt aglet proxy is created and dispatched to the
destination. */
        AgletProxy proxy = getAgletContext().createAglet(getCodeBase(),
"MMHEncryptDecrypt", args);
        proxy = proxy.dispatch(destination);
/* Message is sent to the aglet proxy at the destination and the result is
collected. */
        Message myResult = new Message("result");
        BigInteger Result = (BigInteger)proxy.sendMessage(myResult);
        System.out.println("The returned encrypted result is"+Result);
/*The collected result is decrypted and printed to the console  */
        BigInteger decryptedResult = Result.mod(p);
        System.out.println("The        final        decrypted        result
is"+decryptedResult);
/* The aglet proxy is retracted from the destination and disposed  */
   proxy = getAgletContext().retractAglet(destination,proxy.getAgletID());
   proxy.dispose();
            }catch(MalformedURLException e) {
             }catch(Exception e) {
             }finally {
             }
        }
    }
```

In the next section, we discuss about our second implementation, the secure dynamic programming protocol.

Table 2. Proxy aglet

```
/* This proxy aglet does the multiplication of the two encrypted integers */
public class MMHEncryptDecrypt extends Aglet {
        boolean retracted = false;
        BigInteger encryptedResult ;
/* handles the transfer of the result to the aglet that dispatched this aglet*/
        public boolean handleMessage(Message msg){
                if(msg.sameKind("result")){
                        msg.sendReply(encryptedResult);
                        return true;
```

```
                    } else
                              return false;
                    }
/* onCreation is called when the aglet is created and the arguments(args) are
paased from the creator aglet. */
          public void onCreation(final Object  args) {
               addMobilityListener(
                    new MobilityAdapter() {
/* args values are assigned to Local variables */
          BigInteger encryptedX1 = (BigInteger)((Object[ ])args)[0];
          BigInteger encryptedX2 = (BigInteger)((Object[ ])args)[1];
          BigInteger n =  (BigInteger)((Object[ ])args)[2];
  /* This function is called both the times when the aglet arrives at the
destination(retracted = false) as well as when it is retracted  (retracted =
true)back by the creator. */
          public void onArrival(MobilityEvent e) {
               try {
                    if(retracted) {
          System.out.println("encryptedResult is "+encryptedResult);
                    }else {
                         try{
           encryptedResult = (encryptedX1.multiply(encryptedX2)).mod(n);

}catch (Exception m){
                         }
                    }//end of if-else
               }catch (Exception me) {
                    dispose();
               }
          }// end of onArrival
          public void onReverting(MobilityEvent e) {
                    retracted = true;
          }
     }//end of Mobility adapter
   ); //end of addMobilityListener
  }//end of onCreation
 }
```

8. Secure Dynamic Programming Protocol that utilizes Homomorphic Encryption

In multi-agent systems, multiple autonomous agents sometimes need to solve a combinatorial optimization problem by using their private information. For example, in a combinatorial auction where multiple goods are auctioned simultaneously, agents need to find a combination of bids for disjoint set of goods, so that the sum of the bidding prices is maximized. The problem is called the winner determination problem and has recently become a very active research field [12, 13, 14, 15].

If there exists a fully trusted agent, e.g., the participants can trust the auctioneer, it is possible to gather all private information relevant to the combinatorial optimization problem at this trusted agent; thus this agent can solve the problem using any available centralized optimization technique.

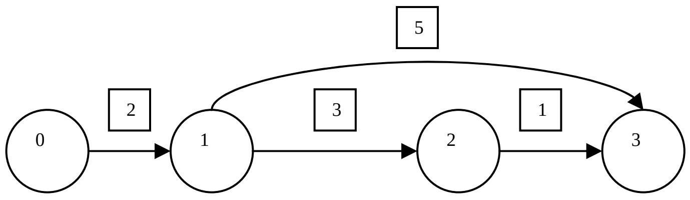
However, we cannot take it for granted that there exists such a trusted agent. For example, in a standard first-price sealed-bid auction [16], where the highest bidder wins and pays his/her own price, the auctioneer might collude with a particular participant and reveal information about incoming bids to that participant during the auction.

We can utilize various cryptographic technologies so that while accepting incoming bids, the auctioneer cannot learn bidding prices. For example, the bidders can first submit encrypted bids, and then give the auctioneer the decryption keys after the bids are closed. However, the auctioneer can utilize the information of bids for future auctions [17].

The proposed solution to this problem is the secure dynamic programming protocol [17] that utilizes indistinguishable, homomorphic and randomizable public key encryption scheme.

An example application of this protocol is the combinatorial auction, where multiple servers can solve a winner determination problem, i.e., they can find the combination of bids so that the sum of the bidding prices is maximized. Although the servers can compute the optimal solution correctly, the information of the bids that are not part of the optimal solution is kept secret even from the servers [17].

Dynamic programming is a powerful method that can be applied to various combinatorial optimization problems. Dynamic programming [16] was developed by R. Bellman during the late 1950's. The Secure dynamic programming [17] protocol is described in the following paragraphs based on the problem of finding the longest path in the one-dimensional directed graph shown in Figure 1. This problem is similar to the winner determination problem described in the previous paragraphs.



Legend:
□ Published weights of the links
○ Nodes or Evaluators

Figure 1. One dimensional directed graph

The graph consists of nodes *0, 1, 2, …, m* with directed links among them. A link is represented as *(j, k)* where $j < k$. For each link *(j, k)*, the weight of the link *w(j, k)* is defined. The goal is to find the longest path from initial node *0* to terminal node *m*., i.e., to find a path from *0* to *m* so that the sum of the weights of links are maximized. For simplicity, we assume for each node *j* (where $0 \leq j < m$), there exists at least one link that starts from *j*, i.e., there is no dead-end node except *m*.

We can obtain the length of the longest path from *0* to *m* by solving the following recurrence formula from node *m - 1* to *0*:

$$f(j) = max_{(j,\ k)}\{w(j, k) + f(k)\}$$

In this formula, *f(j)* represents the length of the longest path from *j* to *m* which is called the evaluation value of node *j*. For terminal node *m*, *f(m)* is defined as *0*. For initial node *0*, *f(0)* represents the optimal solution, i.e., the length of the longest path from *0* to *m*.

The basic idea of the protocol is as follows:

- We assume there is a weight publisher $P_{(j,\ k)}$ for each link *(j, k)*, and an evaluator T_i for each node *i*. In an auction setting, a weight publisher corresponds to a bidder, and an evaluator corresponds to a part of the multiple auction servers.

- These evaluators cooperatively execute *dynamic* programming. Evaluator T_i knows only its evaluation value $f(i)$ and does not know any weight of any link.

The protocol is outlined as follows:

- The weight publisher $P_{(j,k)}$ encrypts its weight $w(j, k)$ using T_j's encryption function.
- Evaluator T_k (who cannot decrypt this information) then calculates the encryption of $w(j,k) + f(k)$.
- Evaluator T_j then calculates $f(j)$ by decrypting a part of this encryption without knowing $w(j, k)$.

To implement this protocol in aglets the encryption scheme we used is the ElGamal encryption scheme which is an indistinguishable, homomorphic and randomizable public key encryption scheme.

Here's how El Gamal works [18]. Pick *a modulo m* (a very large prime number), and two random numbers *b* (the base) and *s* (the secret key) between *1* and *m-1*. Now compute the public key $y = b^s \bmod m$, and publish *m*, *b*, and *y*, keeping *s* secret. Presumably, the difficulty of computing discrete logarithms prevents someone from figuring out *s* from the published information. Now, to send a message *M* (a number between *1* and *m-1*), the sender picks a random number *k* between *1* and *m-1*, and computes:

$$y_1 = b^k \bmod m \text{ and } y_2 = y^k M \bmod m$$

and sends both y_1 and y_2; this is the encrypted message. To decrypt the message requires knowledge of *s*, which allows the following computation:

$$y_1^{-s}\, y_2 \bmod m = b^{-ks}\, b^{ks} M \bmod m = M$$

From the indistinguishability of ElGamal encryption, one can know no information about weight *w* from the encrypted weight *e(w)*.

9. Implementation of Secure Dynamic Programming Protocol in IBM Aglets

The implementation of the protocol is done for the example problem of finding the longest path in the one-dimensional directed graph in Figure 1. This problem and the protocol used to solve the problem are described in detail the previous section. The details of our implementation can be described as follows.

In our implementation there are two evaluator aglets *Evaluator 1 aglet* and *Evaluator 2 aglet* one each for the nodes *1* & *2* respectively. There is also one *Application aglet*. The assumption here is that each one of the evaluators acts honestly and it does not try to decrypt the information that it does not need to know to execute the protocol. The flow of our program is given in Figure 2 and it is described in the following steps.

Step 1: The weight publisher (*Application aglet*) for the weight of the link *(2, 3)* encrypts the weight using ElGamal encryption, which is *e(w(2,3))*. The encrypted weight is then added to *f(3)* (the evaluation value of node 3). Here, *f(3)* is *0* since node *3* is the last node. Then, *e(w(2,3)) + f(3)* is passed as an argument to the evaluator aglet which here is *Evaluator 2 aglet* and is then dispatched to the destination.

Step 2: The *Evaluator 2 aglet* at the destination decrypts *e(w(2,3)) + f(3)* and gets the value of *f(3)*. It then adds this value to *e(w(1, 2))* which was also passed as an argument from the *Application aglet.* The final value *e(w(1,2)) + f(3)* is then passed back to the *Application aglet* at the source by message passing.

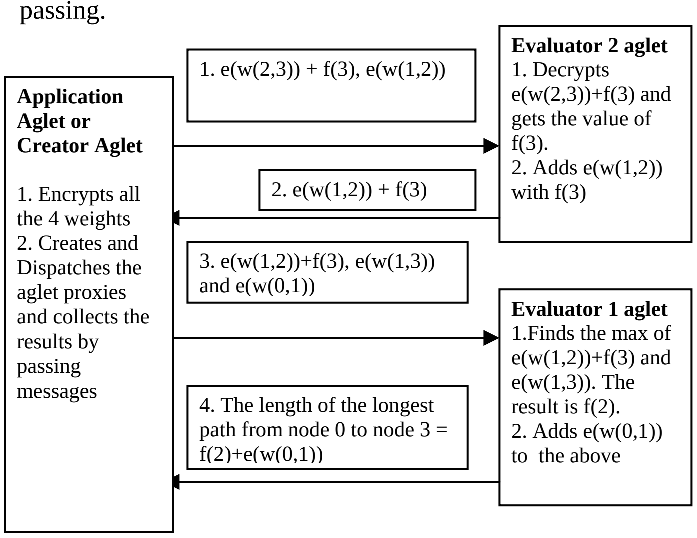

Application Aglet or Creator Aglet
1. Encrypts all the 4 weights
2. Creates and Dispatches the aglet proxies and collects the results by passing messages
1. e(w(2,3)) + f(3), e(w(1,2))
2. e(w(1,2)) + f(3)
3. e(w(1,2))+f(3), e(w(1,3)) and e(w(0,1))
4. The length of the longest path from node 0 to node 3 = f(2)+e(w(0,1))
Evaluator 2 aglet
1. Decrypts e(w(2,3))+f(3) and gets the value of f(3).
2. Adds e(w(1,2)) with f(3)
Evaluator 1 aglet
1.Finds the max of e(w(1,2))+f(3) and e(w(1,3)). The result is f(2).
2. Adds e(w(0,1)) to the above

Figure 2. The program flow of our implementation of Secure Dynamic Programming

Steps 3 and 4 : The *Application aglet* then passes the following values *e(w(1,2))+f(3)*, *e(w(1,3))* and *e(w(0,1))* as arguments to the *Evaluator 1 aglet*. Since the *Evaluator 1 aglet* represents the node *1* and there are two links branching out of node *1*, this aglet has to find out the maximum of the two paths, i.e., the maximum of the following two values: *e(w(1,2))+f(3)* and *e(w(1,3))*. It then adds the value *e(w(0, 1))* to the maximum value. This is the final value which is the length of the longest path, and is now passed back to the *Application aglet* by message passing.

The *Application aglet* then decrypts the final result and prints it to the console.

We represent weight $w\ (1 \le w \le n)$ by encrypted weight *e(w)* that is the following vector of cipher texts [17]:

$$e_{(j,k)}(w(j,k)) = (E_j(z), \ldots, E_j(z), E_j(1), \ldots, E_j(1)).$$

That is, the encrypted weight of the link *(j, k)* is represented by writing the value of $E_j(z)$, $w(j, k)$ times followed by writing the value of $E_j(1)$, $n\text{-}w(j, k)$ times.

It is assumed that *n* is chosen so that it is large enough to represent the length of the longest path. *Z* is the common public element. *Z* is chosen so that $z^k \bmod p \neq 1$ for $0 < k < q$ where q, $p = 2q + 1$ are primes.

Using the above processes, we can find the maximum of weights, and add a constant to a weight without decrypting it. Since we do not reveal the weights to *Evaluator 1* and *Evaluator 2*, they have to perform all the operations on the encrypted weights only, and that is made possible by the vector representation of weights.

Adding a constant to an encrypted weight: We can add a constant such as the evaluation value of a node, to an encrypted weight $e(w) = (e_1, \ldots e_n)$ without decrypting *e(w)* nor learning w. By shifting and randomizing *e(w),* we can obtain

$$e'(w + f) = (E(z), \ldots, E(z), e'_1, \ldots, e'_{n-f})$$

where e'_j is a randomization of ciphertext e_j. Due to randomization one can obtain no information about constant f from $e(w)$ and $e'(w + f)$.

Finding the maximum of two encrypted weights: For example, to find the maximum of $e'(w + f)$ and $e(v)$ (encrypted weight of the weight v), we first create the product of $e'(w + f)$ and $e(v)$ and then start decrypting the elements in the vector representation of this product. We decrypt the elements from last to the first. Every time after decrypting an element, we check to see if it was a *1* or *z*. If it was *z* then the maximum weight is the place number or index number of the element in the vector representation.

In Table 3 the code for the application aglet class file is given. Due to the non-availability of space, the Evaluator aglets' code is omitted here.

Table 3. Application Aglet

```
public class DynaProEncryptionAgletApp extends Aglet {
        static BigInteger [ ][ ] encWeightArray = new BigInteger [6][2];
        static BigInteger [ ][ ] shiftedArray = new BigInteger [6][2];
        static BigInteger [ ][ ] randomizedArray = new BigInteger [6][2];
      static BigInteger [ ][ ] weightPlusConstantArray = new BigInteger [6][2];
        static BigInteger [ ][ ] secondEncWeightArray = new BigInteger [6][2];
        static BigInteger [ ][ ] thirdEncWeightArray = new BigInteger [6][2];
        static BigInteger [ ][ ] MaxWeightArray = new BigInteger [6][2];
        static BigInteger [ ][ ] Result = new BigInteger [6][2];
/** p (modulo) is a very large prime number **/
        static BigInteger p = new BigInteger("23");
/** z is the common public element. Not a variable **/
      static String z = "5";
/** n is the length of the array which is large enough to represent the length of the longest path. Not a variable. */
      static int n = 6;
       public void run() {
/** Variables needed to fill up encWeightArray **/
/** Here w is the weight of the link (2,3). Variable.**/
             int w = 1;
            /** r is the random number chosen for encryption. Variable.**/
             String r = "1";
             /** The constant that is to be added to the encrypted weight **/
             int f = 0;
            /** The function jayElGamel fills up encWeightArray **/
            jayElGamel(w,n,z,r,f);
            /** Variables needed to fill up the secondEncWeight Array
             * Here w is the weight of the link (1,2) */
            w = 3;r = "7";f = 0;
            /**Fills up secondEncWeight array **/
            jayElGamel(w,n,z,r,f);
            /** Variables needed to fill up the thirdEncWeight Array
             * Here w is the weight of the link (0,1) */
            w = 2;r = "13";f = 0;
            /**Fills up thirdEncWeight array **/
            jayElGamel(w,n,z,r,f);
            /** Variables needed to fill up the randomized Array for Evaluator Two **/
             w = 0; r = "9";f = 1;
            /**Fills up randomized array **/
            jayElGamel(w,n,z,r,f);
            try {
            /** Creating an object to pass the arguments to the proxy (Evaluator 2) **/
                   Object args2 = new Object[ ] {
                          encWeightArray,
                          secondEncWeightArray
                    };
                   URL destination2 = new URL("atp://LIFEBOOK:3000");
/ * The EvaluatorTwo aglet proxy2 is created and dispatched to the destination. */
                    AgletProxy proxy2 = getAgletContext().createAglet(getCodeBase(), "EvaluatorTwo", args2);
                    proxy2 = proxy2.dispatch(destination2);
/ * Message is sent to the EvalutorTwo aglet proxy2 at the destination2 and the result is collected.  */
                    Message myResult = new Message("result");
                    Result = (BigInteger[ ][ ])proxy2.sendMessage(myResult);
/ * The collected result is  printed to the console */
                    String tableString = "";
                    for(int i = 0; i < n; i++){
                        for(int j = 0; j < Result[i].length; j++){
                           tableString += Result[i][j]+ "  ";
                        }
                         tableString += "\n";
                    }
                    System.out.println("ResultArray");
                    System.out.println(tableString);
                     decryptWeight(n);
/** Creating an object to pass the arguments to the proxy (Evaluator 1) **/
                     Object args1 = new Object[ ] {
                         encWeightArray,
                         Result,
                         thirdEncWeightArray
                      };
                      URL destination1 = new URL("atp://LIFEBOOK:4000");
            /* The EvaluatorOne aglet proxy1 is created and dispatched to the destination1.*/
                    AgletProxy proxy1 = getAgletContext().createAglet(getCodeBase(), "EvaluatorOne", args1);
                    proxy1 = proxy1.dispatch(destination1);
            /*Message is sent to the EvalutorOne aglet proxy at the destination1 and the result is collected. */
                    Message Result =(BigInteger[ ][ ])proxy1.sendMessage(myResult1);
            /* The collected result is  printed to the console */
                    tableString = "";
                    for(int i = 0; i < n; i++){
                        for(int j = 0; j < Result[i].length; j++){
                               tableString += Result[i][j]+ "  ";
                          }
                    tableString += "\n";
                    }
                    System.out.println("ResultArray");
                    System.out.println(tableString);
                    decryptWeight(n);
            /** The EvaluatorTwo aglet proxy2 is retracted from the destination2 and disposed */
                    proxy2= getAgletContext().retractAglet(destination2,proxy2.getAgletID());
                    proxy2.dispose();
/** The EvaluatorOne aglet proxy1 is retracted from the destination1 and disposed */
                    proxy1= getAgletContext().retractAglet(destination1,proxy1.getAgletID());
                    proxy1.dispose();

           }catch(MalformedURLException e) {
            }catch(Exception e) {
            }finally {
           }
       }
    }
```

10. Checking Mobile Agent's Integrity in IBM Aglets

An adversary can modify the mobile agent's code during their journey from the source to the destination. To avoid this kind of attack, it is always better to check the integrity of the aglet's code once it arrives at the destination.

To achieve this, we have modified the aglet server's source code. This modified server code checks the integrity of the aglets at the destination before they are being executed.

The digest of the agent class file is calculated using *SHA-1* digest algorithm before it is dispatched to the destination and sent along with the agent to the destination. Upon arrival at the destination, the digest of the agent class file is calculated once again and compared with the digest that has been dispatched with the agent. If these two digest values are different then an error message is displayed.

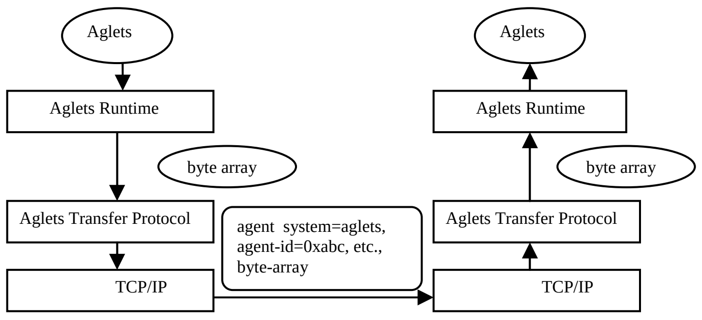


Figure 3. Aglet's architecture

In IBM Aglets, the aglet's class file code is passed between the Aglets Runtime and Aglets Transfer Protocol in a byte array as shown in Figure 3. We used these byte arrays to calculate our digest both at the sending end and at the receiving end.

At the sending end the digest is calculated as shown in Table 4:

Table 4. Digest calculation – Sending end**.**

```
byte[] agent = writer.getBytes();
byte[] agent_digest = new byte[25];
MessageDigest shaTwo =
MessageDigest.getInstance("SHA");
shaTwo.update(agent);
agent_digest = shaTwo.digest();
 System.out.println("LocalAgletRef: SHA Digest Length:"
       + agent_digest.length);
 for(int i=0; i<agent_digest.length;i++){
     System.out.print(agent_digest[i] + " ");
  }
```

At the receiving end the digest is recalculated and verified as shown in Table 5:

Table 5. Digest calculation – Receiving end.

```
MessageDigest shaTwo = MessageDigest.getInstance("SHA");
shaTwo.update(agent);
byte[] hashTwo = shaTwo.digest();
System.out.println("AgletContextImpl: SHA Digest Length:" +
hashTwo.length);
for(int i=0; i<hashTwo.length;i++){
      System.out.print(hashTwo[i] + " ");
}
boolean digestsequal = Arrays.equals(hashTwo, received_agent_digest);
if(digestsequal){
      System.out.println("AgletContextImpl: No Integrity attack");
}else {
     System.out.println("AgletContextImpl: There is an Integrity attack");
}
```

11.Conclusions

In this paper we have shown the implementation of two different security approaches to protect the mobile agents against the malicious hosts, in IBM Aglets. We have also presented our implementation that checks the integrity of the aglets. The security approaches that are implemented are *Mixed-Multiplicative Homomorphic encryption scheme* and *Secure Dynamic Programming*. In these security approaches the computation is done on the encrypted data itself without decrypting, thus providing security. The encryption schemes used in these approaches are Mixed-multiplicative Homomorphic Encryption scheme and ElGamal Encryption Algorithm. Our implementation to check the integrity of the aglets uses SHA-1 digest algorithm. In this way, we have achieved our goal of providing security and integrity to IBM's mobile Agents – Aglets.

References

[1] Danny B. Lange / Mitsuru Oshima, Programming and Deploying JAVA Mobile Agents with Aglets.

[2] Wayne Jansen, Tom Karygiannis. NIST Special Publication 800-19 - Mobile Agent Security. National Institute of Standards and Technology, Computer Security Division, Gaithersburg, MD 20889. {jansen,k arygiannis}@nist.gov.

[3] Fritz Hohl, Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts, in Vigna, Giovanni (Ed.): Mobile Agents and Security, Springer-verlag, 1998.

[4] Gunter Karjoth, Danny B. Lange and Mitsuru Oshima. A Security Model for Aglets. IEEE Internet Computing. Publication date: July 1997. pp. 68-77.

[5] Tomas Sander and Christian F. Tschudin, Towards Mobile Cryptography. Technical Report 97-049, International Computer Science Institute, Berkeley. 1997. http://www.icsi.berkeley.edu/~sander/publications/tr-97-049.ps

[6] M. Abadi and J. Feignbaum. Secure circuit evaluation. Journal of Cryptography, 2(1):1-12, 1990.

[7] R. Rivest, L. Adleman, and M. Dertouzos, On data banks and privacy homomorphisms. In Foundations of Secure Computations, pages 169-178. Academic Press, 1978.

[8] Joseph Domingo Ferrer. A New Privacy Homomorphism and Applications. Universitat Rovira I Virgili. E-mail {jdomingo, jherrera} @ etse.urv.es.

[9] Hyungjick Lee and Jim Alves-Foss and Scott Harrison, The use of Encrypted functions for Mobile Agent Security from the Proceedings of the 37th Hawaii International Conference on System Sciences 2004

[10] Tomas Sander and Christian F. Tschudin, Protecting Mobile Agents Against Malicious Hosts, in Vigna, Giovanni (Ed.): Mobile Agents and Security, Springer-verlag, 1998.

[11] T.Sander and C. Tschudin. On software protection via function hiding. In Information hiding, pages 111-123, 1998.

[12] Y. Fujishima, K. Leyton-Brown, and Y. Shoham.Taming the computation complexity of combinatorial auctions: Optimal and approximate approaches. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99),* pages 548–553, 1999.

[13] M. H. Rothkopf, A. Pekeˇc, and R. M. Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8):1131–1147, 1998.

[14] Y. Sakurai, M. Yokoo, and K. Kamei. An efficient approximate algorithm for winner determination in combinatorial auctions. In Proceedings of the Second ACM Conference on Electronic Commerce (EC-00), pages 30–37, 2000.

[15] T. Sandholm. An algorithm for optimal winner determination in combinatorial auction. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), pages 542–547, 1999.

[16] E. Rasmusen. *Games and Information*. Blackwell, 1994.

[17] Makoto Yokoo, Koutarou Suzuki. Secure Multi-agent Dynamic programming based on Homomorphic Encryption and its Application to Combinatorial Auctions.

[18] www.freesoft.org – An Internet Encyclopedia

[19] Jason Weiss**.** Java Cryptography Extensions – Practical Guide for Programmers. Morgan Kaufmann Publishers, 2004.

[20] Aglets Portal website: http://aglets.sourceforge.net/links.html - development of Aglets being continued by open source community.